Project 3: Scalable Service

For this project I have created two Remote java classes – Cache.java and Server.java and a java.rmi.Remote Interface ServerInterface.java.

The Cache.java implements Cloud.DatabaseOps interface and caches the default database values returned by get(). This class has a **defaultDB** object that references the remote object implementing the default database and a HashMap **cacheTable** to store the key-value entries of the DB. The cache constructor receives a ServerLib object as a request parameter, which is used to get the remote object of the default DB.

The set() and transaction() method forward requests to the default DB. The get() method forwards the request to default DB if the value is not already present in the HashMap and updates its HashMap with the response received from the defaultDB.

The Server class can spin as Coordinator, Front-end Server and App-Server. The first time the Server class is run, it runs as the Coordinator and binds itself with the registry run by the Cloud class. (The first run is identified by exploiting the AlreadyBoundException). On launch the coordinator registers itself with the load-balancer and starts one front-end server, one app-server and instantiates the cache and registers it to the registry. Until the app-server is launched or for the first 3000ms (whichever triggers first) the coordinator drops the requests it receives. In this time it also counts the number of requests received by the coordinator. For every 7 requests received it launches 2 new app-servers and or 1 new app-server for every 5 requests received. This check in the first 3000ms identifies high request rate and proactively launches new app-server, saving the boot time of app-servers at a later stage which eventually leads to request drops and timeouts. I arrived at these numbers by trying different values and identifying the suitable combination for best results. Once the first app-server is launched or a time period of 3000ms ends (whichever happens first) the coordinator spins in an infinite while loop.

The coordinator maintains a global request queue **reqQ**, a HashTable **VMTable** of VMIDs and their respective functions and a count of number of running app-servers - **noMServers**, total number of launched app-servers and number of front-end servers.

In the infinite loop the coordinator pulls requests from its queue and updates the global queue. If the queue length is a greater than the number of app-servers by a particular fraction it drops the incoming requests (This fraction is 1.7 times - arrived by trying different numbers and identifying the best output). If the reqQ length is greater than the no of app-servers, it launches a few app-servers accordingly. This number of new app-servers is the difference between the length of the master queue and the total number of app-servers. This scaling out handles increasing request rate. As the increasing queue length indicates higher load than what the server can handle.

The front-end server identifies itself to be a front-end server by checking the VMTable maintained by the coordinator. It constantly pulls requests and updates the master queue.

The app-server identifies itself by checking the VMTable maintained by the coordinator. Once the app-server is running it updates the number of active servers counts (noMServers) maintained by the coordinator. It then spins in an infinite while loop. The app-servers pull requests from the reqQ maintained by the coordinator and process them through the cache. If an app-server does not receive any request for 6000ms it shuts down itself. This helps in scaling down the app-servers when the queue is empty but also waits for a reasonable amount time before scaling down to avoid unnecessary scale-down.